COMP 3000 Operating Systems

Study Session Questions - Answer Key

December 14, 2025

Part 1 - Overview (OSC Ch. 1-2)

1. Which of the following statements are true? Select all that apply. You may also want to justify your choice behind each decision.

   A. Every interrupt necessarily causes a context switch from **one user process to a different user process.**
   FALSE. Interrupts can switch to kernel mode and back to the same process, or handle the interrupt entirely in kernel mode.

   B. Direct Memory Access (DMA) allows a device to read/write main memory **without CPU involvement for each transferred word**, but the CPU is still notified when the transfer completes.
   TRUE. DMA allows devices to transfer data directly to/from memory without CPU involvement for each word. The CPU is notified via interrupt when the transfer completes.

   C. The primary reason to have a separate kernel mode is **performance**, not protection.
   FALSE. Protection is the primary reason for kernel mode, not performance. Kernel mode provides privileged access for security.

   D. On a single-core CPU, it is **impossible to have parallelism but still possible to have concurrency**.
   TRUE. On a single-core CPU, parallelism (simultaneous execution) is impossible, but concurrency (time-sharing, interleaved execution) is possible.

   E. Microkernel designs generally achieve lower syscall latency than monolithic kernels because they move most services to user space.
   FALSE. Microkernels typically have higher syscall latency (on an overall scale) due to IPC overhead between user-space services and the kernel.

   F. In a layered OS design, a higher layer should not directly invoke services of a non-adjacent lower layer.
   TRUE. This is by definition. Layered design enforces strict hierarchy. Higher layers should only access adjacent lower layers.

   Correct answers: B, D, F.

2. Place the steps of interrupt handling in the correct order, from the beginning. (1, 2, 3, 4)

   1. Device controller/software sends interrupt to CPU
   2. CPU finishes ongoing instruction and saves current program state
   3. CPU uses interrupt vector table to find the right interrupt handler
   4. CPU restores its state and resumes normal operation

3. Place the following storage systems in order of relative typical speed, from slowest to fastest. (1, 2, 3, 4)

   1. Hard-disk drives (slowest, mechanical)
   2. Main memory (RAM)
   3. Cache (L1/L2/L3 cache)
   4. Registers (fastest, on-chip)

4. Explain the differences between a hardware interrupt, a trap (system call), and an exception.

   A **hardware interrupt** is an external event generated by a hardware device, such as when I/O completes or a timer expires. Hardware interrupts are asynchronous, meaning they can occur at any time during program execution, independent of the currently executing instruction. When a hardware interrupt occurs, the CPU saves its current state and transfers control to an interrupt handler.

   A **trap (system call)** is a synchronous, intentional software-generated interrupt that a program uses to request operating system services. The program explicitly executes a special instruction (like a system call instruction) to trigger the trap. This is a deliberate action by the program to request services such as file I/O, process creation, or memory allocation.

   An **exception** is a synchronous, unintentional error condition that occurs during instruction execution. Examples include division by zero, page faults, invalid memory access, or illegal instructions. Exceptions are generated by the CPU when it detects an error condition while executing an instruction. Unlike traps, exceptions are not intentional—they represent error conditions that must be handled by the operating system.

5. Consider the following list of actions. Which of the following should be performed by the kernel, and not by user programs? Select all that apply.

   A. reading the value of the program counter (PC).
   FALSE. User programs can read PC (e.g., via debugger, stack traces).

   B. changing the value of the program counter (PC).
   TRUE. Only kernel can change PC (privileged operation).

   C. increasing the size of an address space.
   TRUE. Memory management (changing address space size) is kernel-only.

   D. creating a memory segment that is shared between multiple processes.
   TRUE. Creating shared memory segments requires kernel privileges.

   E. writing to a memory segment that is shared between multiple processes.
   FALSE. Writing to shared memory is allowed if the process has access permissions.

   F. disabling interrupts.
   TRUE. Disabling interrupts is a privileged operation (kernel-only).

   Correct answers: B, C, D, F.

6. Define the terms system bus, device controller, and DMA controller.

   The **system bus** is a communication pathway that connects the CPU, memory, and I/O devices within a computer system. It allows data transfer between these components by providing a shared channel for address, data, and control signals. The system bus enables the CPU to read from and write to memory and to communicate with peripheral devices.

   A **device controller** is a hardware component that manages a specific I/O device, such as a disk drive, keyboard, or network interface. The device controller handles low-level device operations, translates high-level commands into device-specific actions, and provides a standardized interface to the system. It typically contains registers that the CPU can read and write to control the device and check its status.

   A **DMA (Direct Memory Access) controller** is specialized hardware that allows I/O devices to transfer data directly to and from main memory without requiring CPU involvement for

each transferred word. Instead of the CPU moving each byte individually, the DMA controller manages the entire transfer autonomously. The CPU only needs to initiate the transfer and is notified via interrupt when the transfer completes. This significantly improves I/O efficiency by freeing the CPU to perform other tasks during data transfers.

7. Explain the difference between multiprogramming and multiprocessing. Include what each term means and what hardware support is required.

   Multiprogramming refers to running multiple processes on a single CPU by switching between them when one process blocks, such as when waiting for I/O operations to complete. The operating system maintains several processes in memory simultaneously and switches the CPU among them to maximize CPU utilization. When one process is waiting for I/O, the CPU can execute another process that is ready to run. Multiprogramming requires a single CPU, an interrupt mechanism to handle I/O completion and other events, and memory protection to ensure processes cannot interfere with each other's memory spaces.

   Multiprocessing refers to using multiple CPUs or CPU cores to execute processes simultaneously, enabling true parallel execution. In a multiprocessing system, different processes (or threads within processes) can run concurrently on different processors. This requires multiple CPUs or cores, a shared memory system or interconnect to allow processors to access common memory, and cache coherence mechanisms to ensure that when one processor modifies a memory location, other processors see the updated value. Multiprocessing provides better performance for workloads that can be parallelized, as multiple instructions can execute simultaneously rather than being interleaved in time.

Part 2 - Process Management (OSC Ch. 3-5)

8. List the common process states in a typical OS and give at least three pieces of information stored in the process control block (PCB).

   The common process states in a typical operating system are: **New**, where the process is being created; **Ready**, where the process is loaded in memory and waiting to be assigned to a CPU; **Running**, where the process is currently being executed by the CPU; **Waiting (Blocked)**, where the process is waiting for some event such as I/O completion or a resource to become available; and **Terminated**, where the process has finished execution and is being removed from the system.

   The Process Control Block (PCB) stores essential information about each process. At least three pieces of information stored in the PCB include: (1) **Process ID (PID)**, a unique identifier for the process; (2) **Program Counter (PC)**, which indicates the address of the next instruction to be executed; and (3) **CPU registers**, which store the current state of the process's registers when it is not running. Additional information in the PCB includes the process state, memory management information (such as page tables and memory limits), I/O status (including open files and devices), scheduling information (such as priority and CPU time used), and the parent process ID.

9. Which of the following statements are true? **Select all that apply.**

   A. It is possible to have concurrency **but not parallelism**.
   TRUE. Single-core systems have concurrency via time-sharing, but no parallelism.

   B. Threads within the same process access the same **address space**.
   TRUE. Threads within the same process share the address space.

   C. The **Ready** queue contains processes that are waiting for I/O to complete.
   FALSE. Ready queue contains processes ready to run; I/O-waiting processes are in the Wait queue.

   D. In preemptive priority scheduling, starvation of low-priority processes is **impossible**.
   FALSE. Starvation is possible without aging mechanisms.

   E. A race condition occurs when the correctness of a program depends on the relative timing of threads.
   TRUE. This is the definition of a race condition.

   F. A successful context switch **always** requires saving the state of the currently running process and restoring the state of the next process.
   TRUE. Context switch requires saving current and restoring next process state.

   Correct answers: A, B, E, F.

10. What is the use of the return value of `fork()`?

    The return value of `fork()` is used to distinguish between the parent process and the child process, allowing them to execute different code paths. When `fork()` is called, it creates a new child process that is an exact copy of the parent. The function returns different values to each process: it returns the child's process ID (PID) to the parent process as a positive integer, returns 0 to the child process, and returns -1 if the fork operation fails. This design allows the parent and child processes to determine their identity and execute different code paths based on the return value. For example, the parent might wait for the child to complete, while the child might execute a different program using `exec()`.

11. Which of the following is shared between threads of the same process? Select all that apply.

    A. integer and floating point registers
    FALSE. Each thread has its own registers.

    B. program counter
    FALSE. Each thread has its own program counter.

    C. heap memory
    TRUE. Threads share heap memory.

    D. stack memory
    FALSE. Each thread has its own stack memory.

    E. global variables
    TRUE. Threads share global variables.

    F. open files
    TRUE. Threads share open files.

    Correct answers: C, E, F.

12. Under which of the following basic CPU scheduling policies is starvation possible? Select all that apply.

    A. First-Come-First-Served (FCFS)
    FALSE. No starvation (processes run in arrival order).

    B. Shortest-Job-First (SJF)
    TRUE. Starvation possible (long jobs may never run if short jobs keep arriving).

    C. Shortest-Remaining-Time-First (SRTF)
    TRUE. Starvation possible (same as SJF, preemptive version).

    D. Round-Robin (RR)
    FALSE. No starvation (each process gets time quantum).

    Correct answers: B, C.

13. What is *context switching*?

    Context switching is the process by which the operating system saves the current state of a running process or thread and restores the previously saved state of another process or thread so it can resume execution. When a context switch occurs, the operating system saves relevant information (see Q8) to the PCB of the currently running process, then loads the saved state from the PCB of the process that is about to run.

14. What is the distinction between a ready thread and a waiting thread?

    A **ready thread** is a thread that is waiting only for CPU time to be allocated to it. The thread has all the resources it needs to execute, such as memory, files, and other required resources, and can begin running immediately when the scheduler assigns it to a CPU. The only thing preventing a ready thread from running is the availability of CPU time.

    A **waiting (blocked) thread** is a thread that is waiting for some event or condition to occur before it can continue execution. This event might be I/O completion, a semaphore becoming available, a message arriving, etc. Even if CPU time is available, a waiting thread cannot run because it lacks a necessary resource or is waiting for an external event. Once the awaited event occurs, the thread transitions back to the ready state and can be scheduled for execution.

15. Which of the following statements are true? **Select all that apply.**

    A. With a many-to-one threading model (many user threads mapped to one kernel thread), blocking on a single system call may block all user threads in that process.
    TRUE. One kernel thread blocks, all user threads in that process are blocked.

    B. Adding more cores **always** constantly decreases execution time for a multithreaded program, as long as there is no I/O.
    FALSE. Refer to Amhdal's Law.

    C. Round-Robin scheduling is preemptive because it forces a context switch after each time quantum, even if the running process has not blocked or terminated.
    TRUE. Round-Robin is preemptive by definition.

    D. In a pure one-to-one threading model, creating a very large number of user threads may stress kernel resources and harm performance.
    TRUE. Each user thread creates a kernel thread, consuming kernel resources.

    Correct answers: A, C, D.

16. Compare anonymous pipes and message passing for inter-process communication. Name one advantage of each.

    Anonymous pipes provide a unidirectional byte stream, typically between parent and child processes. It is simple and efficient for linear producer-consumer patterns.

    Message passing involves structured messages with an explicit sender/receiver. It's more flexible (bidirectional, multiple recipients), better for distributed systems, and can include metadata.

17. Explain processor affinity. Why might an OS keep a thread on the same CPU when possible?

    Processor affinity refers to the tendency of a process or thread to continue executing on the same CPU or core when possible, rather than being migrated to different processors. The operating system can implement processor affinity by tracking which CPU a thread last ran on and attempting to schedule it on the same CPU again.

    There are several important reasons why an operating system might maintain processor affinity. First, **cache locality** is improved when a thread runs on the same CPU repeatedly. The CPU's cache will likely contain data and instructions from the thread's previous execution, reducing cache misses and improving performance. Second, **TLB (Translation Lookaside Buffer) effectiveness** is enhanced because the TLB entries for the thread's address space remain valid when it runs on the same CPU, avoiding the need to reload page table translations. Third, **reduced migration overhead** is achieved by avoiding the costs associated with moving a thread to a different CPU, including cache and TLB flushes, potential cache misses, and the overhead of updating scheduler data structures. By keeping threads on the same CPU when possible, the operating system can significantly improve performance, especially for CPU-bound workloads.

18. A system uses an SRT (Shortest Remaining Time) scheduler. Among processes with the shortest remaining time, the earliest-arriving one is chosen.

    Three processes arrive: P_0, P_1, and P_2. P_0 has a CPU burst of 4 units, P_1 has a burst of $(1+y)$ units, P_2 has a burst of 3 units.

    The processes arrive at times 0, 2, and 5 (one process at each time).

    1. Find the value of y so that:
        - SRT produces the **same** schedule as FCFS.
        - The average response time of the three processes is **maximized**.

    Response time is the time from arrival to first execution.

    For SRT = FCFS:

    - FCFS order: P_0 (0-4), P_1 (4 to 4+1+y), P_2 (4+1+y to 4+1+y+3)
    - For SRT to match, P_1 must not preempt P_0 at time 2, so $1 + y \geq 4 - 2 = 2$, thus $y \geq 1$
    - Also, P_2 must not preempt P_1, so $3 \geq (1 + y) - (5 - 4) = y$, thus $y \leq 3$

    For maximum response time:

    - Response times: $R_0 = 0$, $R_1 = 4 - 2 = 2$, $R_2 = (4 + 1 + y) - 5 = y$
    - Average $= (0 + 2 + y)/3 = (2 + y)/3$, maximized when y is maximum
    - With $y \geq 1$ and $y \leq 3$, maximum is $y = 3$

    Answer: $y = 3$

19. Consider the following set of processes, with the length of the CPU burst time given in milliseconds:

Process	Burst Time	Priority
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

The processes are assumed to have arrived in the order P_1, P_2, P_3, P_4, P_5, all at time 0.

Which of the following algorithms results in the minimum average waiting time (over all processes)?

A. First-Come-First-Served (FCFS)

B. Shortest-Job-First

C. Non-preemptive Priority (with a larger priority number implying a higher priority)

D. Round-Robin (quantum = 2)

Shortest-Job-First minimizes average waiting time.

Calculations:

FCFS: Order P1(0-2), P2(2-3), P3(3-11), P4(11-15), P5(15-20)

- Wait times: P1=0, P2=2, P3=3, P4=11, P5=15
- Average = 31/5 = 6.2

SJF: Order P2(0-1), P1(1-3), P4(3-7), P5(7-12), P3(12-20)

- Wait times: P1=1, P2=0, P3=12, P4=3, P5=7
- Average = 23/5 = 4.6

Priority: Order P3(0-8), P5(8-13), P1(13-15), P4(15-19), P2(19-20)

- Wait times: P1=13, P2=19, P3=0, P4=15, P5=8
- Average = 55/5 = 11.0

RR (q=2):

- Schedule: P1(0-2, completes), P2(2-3, completes), P3(3-5, remaining=6), P4(5-7, remaining=2), P5(7-9, remaining=3), P3(9-11, remaining=4), P4(11-13, completes), P5(13-15, remaining=1), P3(15-17, remaining=2), P5(17-18, completes), P3(18-20, completes)
- Completion times: P1=2, P2=3, P3=20, P4=13, P5=18
- Turnaround times: P1=2-0=2, P2=3-0=3, P3=20-0=20, P4=13-0=13, P5=18-0=18
- Wait times: P1=0, P2=2, P3=20-8=12, P4=13-4=9, P5=18-5=13
- Average wait time = (0+2+12+9+13)/5 = 36/5 = 7.2

Part 3 - Process Syncronization (OSC Ch. 6-8)

20. Match each term to the most accurate and precise defenition.

A. Mutual Exclusion	**1.** Each process waits at most a finite number of turns.
B. Bounded Waiting	**2.** If the critical section is empty, waiting processes must eventually enter.
C. Progress	**3.** Only one process may be in its critical section.

A → 3, B → 1, C → 2.

- **A. Mutual Exclusion → 3.** Only one process may be in its critical section
- **B. Bounded Waiting → 1.** Each process waits at most a finite number of turns
- **C. Progress → 2.** If the critical section is empty, waiting processes must eventually enter

21. Which of the following statements are true? **Select all that apply.**

A. A binary semaphore can always be used as a mutex, but a mutex cannot always be used as a general counting semaphore.
TRUE. Binary semaphore (0/1) can act as mutex; mutex is binary-only and cannot be used as general counting semaphore.

B. In a readers–writers lock with reader priority, it is possible for writers to starve even when there is only a single writer repeatedly requesting access.
FALSE. With reader priority, readers can starve writers, but a single writer won't starve itself.

C. If a system's resource-allocation graph contains a cycle, the system is definitely in a deadlocked state.
FALSE. Cycle is necessary but not sufficient; need all resources single-instance or cycle with multi-instance resources.

D. Deadlock avoidance (e.g., Banker's algorithm) can sometimes deny a resource request even though granting it would not immediately guarantee deadlock.
TRUE. Banker's algorithm is conservative—denies if unsafe, even if granting would be safe.

E. Deadlock detection algorithms **must** know the maximum resource demand of each process to function correctly.
FALSE. Detection needs current allocation and requests, not necessarily maximum demand.

Correct answers: A, D.

22. (a) How could one PREVENT deadlock in the situation shown below using deadlock prevention techniques discussed in class?

Four buses blocking each other at the Alexander Klellands Plass intersection in Oslo, Norway. Photo dated 24 November 2025.

(a) Deadlock prevention techniques can be applied to this traffic situation in several ways. First, **violating mutual exclusion** would involve allowing multiple buses to share the same lanes simultaneously (though this is obviously not practical). Second, **preventing hold and wait** would require buses to request all needed lanes before entering the intersection, ensuring they have all resources before proceeding. Third, **allowing preemption** would enable a traffic controller to force traffic to back up and release its current position, allowing another bus to proceed. Fourth, **preventing circular wait** would involve establishing a priority ordering (such as north > east > south > west) so that buses always yield to buses coming from higher-priority directions, breaking any potential circular dependencies. The most practical solution is circular wait prevention via traffic rules, which is commonly implemented in real traffic systems through right-of-way rules and traffic signals.

(b) Provide another (non-operating-system-related) example of deadlock.

(b) A common real-world example of deadlock occurs when two people meet in a narrow hallway or doorway, each waiting for the other to move. Both individuals hold their position (which represents a resource) and wait for the other person to release theirs by moving. This creates a circular wait condition where person A is waiting for person B to move, and person B is waiting for person A to move, resulting in a deadlock where neither can proceed. This situation can only be resolved by breaking one of the deadlock conditions, such as one person backing up (preemption) or establishing a rule like "yield to the person on the right" (preventing circular wait).

23. Given the state below, determine whether the system is in a safe state. If it is safe, provide a safe sequence.

Allocation

P	R_1	R_2	R_3
P0	2	4	3
P1	0	1	1
P2	5	0	2

Need

P	R_1	R_2	R_3
P0	3	2	4
P1	1	4	3
P2	2	0	0

Available

R_1	R_2	R_3
2	1	2

Given the allocation matrix showing P0=(2,4,3), P1=(0,1,1), P2=(5,0,2), the need matrix showing P0=(3,2,4), P1=(1,4,3), P2=(2,0,0), and available resources (2,1,2), we need to determine if the system is in a safe state.

First, we calculate the maximum demand (Max) for each process by adding Allocation + Need. For P0: (2+3, 4+2, 3+4) = (5,6,7). For P1: (0+1, 1+4, 1+3) = (1,5,4). For P2: (5+2, 0+0, 2+0) = (7,0,2).

To check if the system is safe, we look for a safe sequence—an ordering of processes such that each process can complete with the resources currently available plus those released by previously completed processes. Starting with Available = (2,1,2), we check if any process's Need can be satisfied. P2's Need (2,0,0) is less than or equal to Available (2,1,2), so P2 can complete. If P2 completes, it releases its allocation, making Available = (2,1,2) + (5,0,2) = (7,1,4).

However, with Available = (7,1,4), neither P0 nor P1 can complete. P0's Need (3,2,4) requires 2 units of R2, but only 1 is available. P1's Need (1,4,3) requires 4 units of R2, but only 1 is available. Since we cannot find a safe sequence where all processes can eventually complete, **the system is UNSAFE**.

24. Consider the following information about resources in a system:

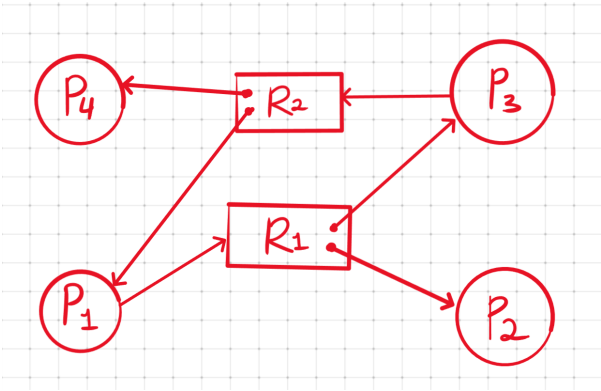
- There are two classes of allocatable resource labelled R1 and R2.
- There are two instances of each resource.
- There are four processes labelled P1 through P4.
- There are some resource instances already allocated to processes, as follows:
  - one instance of R1 held by P2, another held by P3
  - one instance of R2 held by P1, another held by P4
- Some processes have requested additional resources, as follows:
  - P1 wants one instance of R1
  - P3 wants one instance of R2

(a) Draw the resource allocation graph for this system.

(b) What is the state (runnable, waiting) of each process? For each process that is waiting, indicate what it is waiting for.

(c) Is this system deadlocked? If so, state which processes are involved. If not, give an execution sequence that eventually ends, showing resource acquisition and release at each step.

(a) See figure below.



(b) P1 is in a waiting state because it holds one instance of R2 but wants one instance of R1, which is currently fully allocated (both instances held by P2 and P3). P2 is in a runnable state because it holds one instance of R1 and has no outstanding resource requests, so it can continue execution. P3 is in a waiting state because it holds one instance of R1 but wants one instance of R2, which is currently fully allocated (both instances held by P1 and P4). P4 is in a runnable state because it holds one instance of R2 and has no outstanding resource requests.

(c) The system is **NOT deadlocked**. Although P1 and P3 are waiting and form a potential cycle (P1 holds R2 and wants R1, while P3 holds R1 and wants R2), the deadlock can be broken because P2 and P4 are runnable and can complete their execution. If P2 completes and releases its instance of R1, then P1 can acquire R1, complete its execution, and release R2. Once R2 is released, P3 can acquire it and complete. Finally, P4 can complete. Alternatively, if P4 completes first and releases R2, then P3 can acquire R2, complete, and release R1, allowing P1 to proceed. Since there exists an execution sequence where all processes can eventually complete, the system is not deadlocked.

25. In Peterson's solution for two processes:

```
do {
  flag[i] = true;
  turn = j;
  while (flag[j] && turn == j)
      ; // busy wait
  // critical section
  ...
  flag[i] = false;
  // remainder section
} while (true);
```

Explain how this ensures mutual exclusion and bounded waiting.

Mutual exclusion is ensured through the combination of the `flag` array and the `turn` variable. When process i wants to enter its critical section, it first sets `flag[i]=true` to indicate its interest, then sets `turn=j` to give the other process priority. Process i then waits in a busy-wait loop while `flag[j] && turn==j` is true. If process j is in its critical section, `flag[j]` will be true. If both processes try to enter simultaneously, the `turn` variable determines which process proceeds—only one process can have `turn` set to its own index at any given time. The waiting condition ensures that if both processes are interested, only the one indicated by `turn` can proceed, while the other must wait. This guarantees that at most one process can be in its critical section at any time.

Bounded waiting is ensured because when process j exits its critical section, it sets `flag[j]=false`. If process i was waiting, it will eventually be able to proceed. This happens because either (1) `turn` was already set to i when process j last entered (since process j sets `turn=i` before entering), or (2) process j will set `turn=i` on its next attempt to enter the critical section. Since process j must set `turn=i` before entering, and process i will eventually see `flag[j]=false` or `turn==i`, process i is guaranteed to enter within at most one turn of process j. This ensures that no process can be starved and that waiting is bounded.

26. In the classic bounded producer–consumer problem with one buffer of size N:

- mutex protects the buffer.
- empty is a counting semaphore initialized to N.
- full is a counting semaphore initialized to 0.

(a) Why do producers call wait(empty) before inserting an item?
(b) Why is it incorrect to remove mutex and only rely on empty/full?

(a) Producers call `wait(empty)` before inserting an item to ensure that there is available space in the buffer before attempting to insert data. The `empty` semaphore is initialized to N (the buffer size) and tracks the number of empty slots available. When a producer calls `wait(empty)`, it decrements the semaphore. If the buffer is full (meaning `empty` has value 0), the `wait(empty)` operation will block the producer until a consumer removes an item and calls `signal(empty)`. This mechanism prevents buffer overflow by ensuring that producers cannot insert items into a full buffer, maintaining the bounded buffer constraint.

(b) It is incorrect to remove mutex and only rely on `empty` and `full` because these semaphores only track the number of available buffer slots, not the integrity of the buffer data structure itself. Without the mutex, multiple producers could simultaneously execute the following sequence: (1) check that `empty > 0`, (2) decrement `empty`, and (3) write to the same buffer slot, causing data corruption as they overwrite each other's data. Similarly, multiple consumers could read from the same slot, or a consumer could read data that a producer is still writing (partially written data). The mutex ensures that only one process (producer or consumer) can access the buffer structure at a time, making the check-and-modify operations on the buffer atomic. This prevents race conditions and ensures data integrity in the shared buffer.

Part 4 - Memory Management (OSC Ch. 9-10)

27. Which of the following statements are true? **Select all that apply.**

    A. Paging eliminates both internal and external fragmentation.
    FALSE. Paging eliminates external fragmentation but can have internal fragmentation (last page partially empty).

    B. Multi-level paging trades higher address translation overhead for lower page-table memory usage.
    TRUE. Multi-level reduces page table size but requires multiple memory accesses for translation.

    C. LRU page replacement can be approximated using a single reference bit per page and a circular scan (clock algorithm).
    TRUE. Clock algorithm approximates LRU using reference bit.

    D. If a process's working set does not fit into memory, increasing the degree of multiprogramming will usually reduce thrashing/the # of page faults.
    FALSE. Increasing multiprogramming when working sets don't fit increases thrashing (more processes compete for limited frames).

    Correct answers: B, C.

28. Under what conditions might two or more processes share common physical frames? Explain how and why this would be done.

    Two or more processes can share common physical frames under several conditions, each serving different purposes. First, **shared libraries and code pages** can be mapped to the same physical frames when multiple processes load the same read-only library code, such as the C standard library (libc). Since these pages contain executable code that never changes, they can be safely shared across processes. The operating system maps the virtual addresses of different processes to the same physical frames, significantly saving memory by avoiding duplicate copies of common libraries.

    Second, **shared memory** is an inter-process communication (IPC) mechanism where processes explicitly request to share memory regions through system calls like `shmget()` or `mmap()`. The operating system allocates physical frames and maps them into the virtual address spaces of multiple processes. This allows processes to communicate by reading and writing to the shared memory region, providing efficient data sharing without the overhead of message passing or pipes.

    Third, **copy-on-write (COW)** allows parent and child processes created via `fork()` to initially share the same physical frames for their address spaces. Both processes' page tables point to the same physical frames. However, these pages are marked as read-only. When either process attempts to write to a shared page, a page fault occurs, and the operating system creates a private copy of that page for the writing process. This mechanism saves memory during process creation since most pages are never modified, and only pages that are actually written need to be copied.

29. If every page in memory is accessed (used) between any two page faults, will the Clock replacement algorithm behave exatcly like FIFO? Justify your response whether true or false.

    True. The Clock replacement algorithm will behave exactly like FIFO if every page in memory is accessed between any two page faults.

The Clock algorithm uses a reference bit for each page frame. When a page is referenced (accessed), its reference bit is set to 1. The algorithm maintains a circular list of pages and a "clock hand" that scans through them. When a page replacement is needed, the clock hand scans the pages, clearing reference bits as it goes, and evicts the first page it encounters with a reference bit of 0.

If every page is accessed between any two page faults, all reference bits will be set to 1 when a page fault occurs and the clock hand begins its scan. As the clock hand scans, it clears the reference bits of pages it passes (setting them to 0). Since all pages had their bits set to 1, the clock hand will clear bits in circular order until it returns to the first page it encountered, which will now have bit=0 (because it was cleared during the scan). This page will be evicted.

The key insight is that when all pages are accessed, Clock's circular scan order matches FIFO's eviction order. The page that was first in the circular list (the oldest page in memory, assuming pages were loaded in order) will be the first one the clock hand encounters and clears, making it the first candidate for eviction. Since Clock maintains its position between scans, and all pages are accessed (making their behavior identical), Clock effectively evicts pages in the same order they entered memory, which is exactly how FIFO works. Therefore, Clock behaves like FIFO under this condition.

30. Assume that there are 5 pages, A, B, C, D, and E. Fill in the page reference string and complete the rest of the information in the table below so that LRU is the *worst* page replacement algorithm (i.e., it results in the maximum number of page faults). Use a dash "–" to fill in blank locations. Note that when there is more than one page that is a possible victim, always choose the one with the **lowest frame number**.

Num	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Refs	A	B	C	D	E										
Frame 1	A														
Frame 2	–														
Frame 3	–														
Frame 4	–														
Fault?	X														

To maximize the number of page faults for LRU, we need to force LRU to evict pages that will be needed immediately in the future. The strategy is to fill all available frames, then reference pages in reverse order of their recency, ensuring that LRU always evicts the page that will be needed next.

The reference string that achieves this is: A, B, C, D, E, A, B, C, D, E, A, B, C, D, E (a repeating pattern). With 4 frames available, the first 4 references (A, B, C, D) will cause page faults as frames are filled. When E is referenced, LRU will evict A (the least recently used). Then when A is referenced again, it causes a fault and evicts B (now the least recently used). This pattern continues, causing LRU to fault on every single reference after the initial 4, resulting in 15 total page faults.

This makes LRU the worst algorithm for this pattern because it always evicts the least recently used page, which in this carefully constructed sequence is always the page that will be needed next. The algorithm's strength (using recency information) becomes its weakness when the access pattern is designed to exploit it.

31. Consider a virtual memory system that uses 2-level paging. The page size in this system is 256 (2^8) bytes. Each individual page table fits exactly into one memory frame, and the size of each page table entry (PTE) is 8 bytes.

    (a) What is the maximum size (in bytes) of a virtual address space in this system?

    To determine the maximum virtual address space size, we need to calculate how many pages can be addressed with the 2-level paging structure. First, the page size is 256 bytes, which equals 2^8 bytes, so the page offset portion of the virtual address requires 8 bits.

    Second, each page table must fit exactly into one memory frame, which is 256 bytes. Since each page table entry (PTE) is 8 bytes, each page table can contain $256/8 = 32$ entries, which equals 2^5 entries.

    Third, with 2-level paging, the outer page table has 32 entries, and each of these entries points to an inner page table that also has 32 entries. Therefore, the total number of pages that can be addressed is $32 \times 32 = 1024 = 2^{10}$ pages.

    Finally, the maximum virtual address space size is calculated by multiplying the number of addressable pages by the page size: $2^{10} \times 2^8 = 2^{18} = 262,144$ bytes, which equals 256 KB.

    (b) Suppose that there is a process with a virtual address space of the maximum size. How many bytes of memory are occupied by the page tables for this process?

    For a process with the maximum virtual address space size, we need to account for all page tables required. The maximum size uses all 1024 possible pages. To address these pages, we need one outer page table, which occupies 256 bytes (one frame). Additionally, we need 32 inner page tables, one for each entry in the outer page table. Each inner page table occupies 256 bytes, so the total for all inner page tables is $32 \times 256 = 8192$ bytes.

    The total memory occupied by page tables is therefore $256 + 8192 = 8448$ bytes. Note that even if not all pages are actually valid (present in memory), the maximum addressable address space requires all inner page tables to exist, as the page table structure must be able to map to any virtual address within the maximum range.

    (c) Suppose that there is a process with a virtual address space of size 10240 ($10 \cdot 2^10$) bytes. Also suppose that the entire address space is in memory. How many valid PTEs will exist in the page tables for this process?

    For a process with a virtual address space of 10240 bytes, we first calculate how many pages this represents. Dividing the address space size by the page size: $10240/256 = 40$ pages. Since the entire address space is in memory, all 40 pages are valid and present.

    Each page requires exactly one page table entry (PTE) to map it to a physical frame. Therefore, there will be 40 valid PTEs in the page tables for this process. Note that the system may need to allocate multiple inner page tables to hold these 40 PTEs (since each inner page table can hold 32 entries), but only 40 of the PTEs will be marked as valid/present, corresponding to the 40 pages that are actually in use.

32. Assume that a program has just referenced an address in virtual memory. Describe a scenario in which each of the following can occur. (If no such scenario can occur, explain why.)

    (a) TLB miss with no page fault

    A TLB miss with no page fault occurs when the page being accessed is already present in physical memory, but the translation from virtual address to physical address is not currently cached in the Translation Lookaside Buffer. When a TLB miss occurs, the memory management

unit (MMU) performs a page table walk to find the page table entry (PTE). If the PTE is found and marked as valid (present bit set), the translation is loaded into the TLB, and the memory access proceeds normally. No page fault occurs because the page is already in physical memory—only the translation was missing from the TLB cache. This is a common scenario when the TLB is full and older entries have been evicted, or when accessing a page for the first time after a context switch.

(b) TLB miss with page fault

A TLB miss with a page fault occurs when the page being accessed is not currently in physical memory. When a TLB miss occurs, the MMU performs a page table walk to find the PTE. During this walk, the MMU discovers that the PTE is marked as invalid (present bit is clear), indicating that the page has been paged out to secondary storage. This triggers a page fault. The operating system's page fault handler then loads the required page from disk into physical memory, updates the PTE to mark it as valid and set the physical frame number, and then loads the translation into the TLB. After the page fault is handled, the original memory access can proceed.

(c) TLB hit with no page fault

A TLB hit with no page fault is the most common and efficient case for memory access. This occurs when the translation from virtual address to physical address is found in the TLB, and the corresponding page is present in physical memory. The MMU uses the cached translation directly from the TLB to access the physical memory location, requiring no page table walk and no page fault handling. This is the fastest possible memory access scenario, as it avoids both the overhead of walking page tables and the much larger overhead of handling a page fault.

(d) TLB hit with page fault

This scenario cannot occur. A TLB hit means that a valid translation was found in the TLB, which implies that the page table entry (PTE) was valid and marked as present when the TLB entry was created. If a page is paged out to disk, the operating system invalidates the corresponding TLB entry (either by clearing it or marking it as invalid) to ensure consistency. Therefore, if the TLB contains an entry for a virtual address, it means the page must be in physical memory. A TLB hit with a page fault is logically impossible because the presence of a TLB entry guarantees that the page is present in memory. If a page fault were to occur, it would mean the page is not in memory, which contradicts the existence of a valid TLB entry.

Part 5 - Storage Management (OSC Ch. 11-12)

33. Which of the following statements are true? **Select all that apply.**

    A. SCAN disk scheduling guarantees less total head movement than FCFS for every possible request pattern.
    FALSE. SCAN doesn't guarantee less movement for all patterns (e.g., requests all at one end).

    B. SSD/NVM devices have smaller benefit from seek-optimizing algorithms than HDDs because they have no mechanical heads to move.
    TRUE. SSDs have no seek time, so seek optimization less beneficial.

    C. For HDDs, logical block addresses (LBAs) are conceptually arranged in order across the disk, even if the controller hides the exact physical layout.
    TRUE. LBAs map sequentially even if physical layout differs.

    D. An OS can improve swap performance by spreading swap space across multiple devices instead of putting all swap on a single disk.
    TRUE. Parallel access to multiple swap devices improves throughput.

    Correct answers: B, C, D.

34. Match each I/O style to the most accurate definition.

A. Blocking I/O	**1.** A single system call reads from or writes to multiple user buffers in one operation.
B. Non-blocking I/O	**2.** The system call does not return until the requested I/O has completed or an error occurs.
C. Asynchronous I/O	**3.** The kernel starts the I/O and returns immediately; completion is reported later (e.g., via signal, callback, or completion queue) so the caller does not need to poll.
D. Vectored I/O	**4.** The system call returns immediately; if the operation would block, it fails (e.g., with "would block"), and the caller must retry or use readiness notification.

    A → 2, B → 4, C → 3, D → 1.

    - **A. Blocking I/O → 2.** The system call does not return until the requested I/O has completed or an error occurs.
    - **B. Non-blocking I/O → 4.** The system call returns immediately; if the operation would block, it fails (e.g., with "would block"), and the caller must retry or use readiness notification.
    - **C. Asynchronous I/O → 3.** The kernel starts the I/O and returns immediately; completion is reported later (e.g., via signal, callback, or completion queue) so the caller does not need to poll.
    - **D. Vectored I/O → 1.** A single system call reads from or writes to multiple user buffers in one operation.

35. Suppose that a disk drive has 8,000 cylinders, numbered from 0 to 7,999. The drive is currently serving a request at cylinder 3,400, and the previous request was at cylinder 3,000. The queue of pending requests, in FIFO order, is:

$$120,\ 3{,}760,\ 1{,}550,\ 720,\ 4{,}090,\ 2{,}780,\ 510,\ 3{,}900.$$

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

(a) FCFS

(b) SCAN

(c) C-SCAN

Given: Current position = 3400, Previous = 3000 (moving upward), Queue: 120, 3760, 1550, 720, 4090, 2780, 510, 3900

FCFS: Order: $3400 \to 120 \to 3760 \to 1550 \to 720 \to 4090 \to 2780 \to 510 \to 3900$

$$\begin{aligned}\text{Distance} &= |3400-120| + |120-3760| + |3760-1550| + |1550-720| \\ &\quad + |720-4090| + |4090-2780| + |2780-510| + |510-3900| \\ &= 3280 + 3640 + 2210 + 830 + 3370 + 1310 + 2270 + 3390 \\ &= 20,300 \text{ cylinders}\end{aligned}$$

SCAN: Head moving upward. Order: $3400 \to 3760 \to 3900 \to 4090 \to 7999$ (end) $\to 2780 \to 1550 \to 720 \to 510 \to 120 \to 0$

$$\begin{aligned}\text{Distance} &= |3400-3760| + |3760-3900| + |3900-4090| + |4090-7999| \\ &\quad + |7999-2780| + |2780-1550| + |1550-720| + |720-510| \\ &\quad + |510-120| + |120-0| \\ &= 360 + 140 + 190 + 3909 + 5219 + 1230 + 830 + 210 + 390 + 120 \\ &= 12,598 \text{ cylinders}\end{aligned}$$

C-SCAN: Head moving upward. Order: $3400 \to 3760 \to 3900 \to 4090 \to 7999 \to 0 \to 120 \to 510 \to 720 \to 1550 \to 2780$

$$\begin{aligned}\text{Distance} &= |3400-3760| + |3760-3900| + |3900-4090| + |4090-7999| \\ &\quad + |7999-0| + |0-120| + |120-510| + |510-720| + |720-1550| \\ &\quad + |1550-2780| \\ &= 360 + 140 + 190 + 3909 + 7999 + 120 + 390 + 210 + 830 + 1230 \\ &= 15,378 \text{ cylinders}\end{aligned}$$

36. Which of the following statements are true? **Select all that apply.**

    A. I/O protection is enforced partly by marking I/O instructions as privileged and by protecting memory-mapped device registers from direct user access.
    TRUE. I/O protection via privileged instructions and memory protection.

    B. Spooling is used when a device can handle multiple concurrent requests in parallel, so the kernel can overlap operations from different processes.
    FALSE. Spooling is for sequential devices (like printers) that can't handle concurrent requests—requests queued.

    C. The kernel's error handling may retry transient I/O errors and can decide to stop using a device that exhibits too many correctable errors.
    TRUE. Kernel retries transient errors, may disable device after repeated failures.

    D. The kernel's device-status table tracks the state of each I/O device so drivers and the kernel can coordinate ongoing operations.
    TRUE. Device-status table tracks device state for coordination.

    Correct answers: A, C, D.

37. Consider the following I/O scenarios on a multitasking workstation:

    (a) A USB keyboard used with a text editor and graphical terminal.

    (b) A network printer shared by many users, connected over Ethernet.

    (c) An SSD that stores the operating system and user home directories.

    (d) An audio output device (sound card) playing compressed music or video sound, using DMA.

    For each scenario, would you design the operating system to use buffering, spooling, caching, or some combination?

    Briefly justify your choices.

    (a) USB keyboard: The operating system should use **buffering** only for a USB keyboard. Keyboard input arrives character-by-character as the user types, and buffering is necessary to handle the mismatch between the variable typing speed of the user and the processing speed of the system. The buffer collects keystrokes so that the system can process them in batches rather than handling each keystroke individually. No spooling is needed because a keyboard is an input device, not an output device that requires queuing.

    (b) Network printer: The operating system should use **spooling** primarily, plus buffering for a network printer. A printer is a slow, sequential output device that can only handle one print job at a time. Spooling allows multiple users to submit print jobs immediately by storing them in a queue (spool), while the printer processes them one at a time. This prevents users from having to wait for the printer to become available. Buffering is also needed to handle network packet assembly, as print data arrives in network packets that must be reassembled into complete print jobs before being sent to the printer.

    (c) SSD storage: The operating system should use **caching** primarily, plus buffering for an SSD that stores the operating system and user files. Caching is essential because frequently accessed files and blocks can be kept in memory, dramatically reducing the number of disk I/O operations and improving performance. Since SSDs are faster than traditional hard drives but still slower than RAM, caching provides significant benefits. Buffering is used for write

operations to batch multiple small writes together, reducing the number of write operations to the SSD and improving write efficiency.

(d) Audio output with DMA: The operating system should use **buffering** primarily for an audio output device playing compressed music or video sound. Audio playback requires a continuous, uninterrupted stream of data to prevent audio dropouts or glitches. Buffering ensures smooth playback by maintaining a queue of audio data ahead of the current playback position, so that even if there are temporary delays in reading or decompressing data, playback can continue smoothly. DMA handles the actual transfer of audio data to the sound card, but buffering prevents underruns (when the buffer runs empty)..

Part 6 - File System (OSC Ch. 13-14)

38. Which of the following statements are true? **Select all that apply.**

    A. Sequential file access requires the hardware to support sequential-only disks; the OS cannot simulate sequential access on random-access devices.
    FALSE. OS can simulate sequential access on random-access devices.

    B. Direct (random) access to a file is commonly implemented using relative block numbers so the OS can choose where to place blocks on disk.
    TRUE. Relative block numbers allow OS flexibility in block placement.

    C. In a two-level directory scheme, all users share a single directory without separation, which avoids name collisions between users and supports hierarchical grouping of their files.
    FALSE. Two-level has user directories, providing separation.

    D. The OS understands the meaning of file extensions like `.c` or `.docx` and uses them to decide how to interpret file contents.
    FALSE. OS doesn't interpret extensions; applications do.

    E. In an acyclic-graph directory, a file or subdirectory may have multiple names (aliases), which can lead to dangling references if one alias is deleted.
    TRUE. Multiple names (links) can create dangling references.

    F. General graph directory structures must use additional mechanisms, such as forbidding links to directories or running cycle detection, to avoid cycles.
    TRUE. General graphs need cycle prevention mechanisms.

    Correct answers: B, E, F.

39. Explain how directory structures (flat, two-level, tree, acyclic, graph) affect usability, protection, and sharing. Give an example scenario where each structure would be particularly convenient or problematic.

    Flat directory structure uses a single directory containing all files in the system. This structure is simple to implement and understand, but provides no organization or hierarchy. All users see all files, which creates poor protection and sharing control—there is no way to restrict access to specific files or organize files by user or purpose. This structure was used in early simple operating systems but is problematic for multi-user environments where users need privacy and the ability to organize their files.

    Two-level directory structure provides a master directory containing user directories, with each user having their own directory. This improves protection by isolating users' files from each other, and provides basic sharing control. However, organization is still limited since users cannot create subdirectories. This structure was common in early time-sharing systems but is problematic for complex file organization needs, as users cannot group related files into subdirectories.

    Tree (hierarchical) directory structure organizes files in a tree with a root directory and subdirectories, similar to a family tree. This provides excellent organization, allowing users to group related files logically. Protection is implemented through file permissions that can be set on directories and files. Sharing is accomplished through explicit paths, though it requires knowing the exact path to shared files. This structure is used in modern Unix/Linux systems

and is convenient for most uses. However, it is problematic if you need multiple paths to access the same file, as each file can only have one parent directory.

Acyclic-graph directory structure extends the tree structure by allowing links (aliases), enabling files and directories to have multiple parent directories. This provides better sharing capabilities, as the same file can be accessed via different paths, making it convenient for shared libraries and common files. However, this structure can create dangling references if a link target is deleted while links to it still exist. Unix systems with symbolic links use this structure, which is convenient for shared libraries but problematic if links are not properly maintained when files are moved or deleted.

General graph directory structure allows full linking including cycles, providing maximum flexibility. However, cycles can create infinite loops when traversing directories and require cycle detection algorithms. This structure is rarely used in practice because the complexity of managing cycles and preventing infinite loops outweighs the benefits. It is problematic due to the complexity of implementation and the potential for confusing directory traversal.

40. Explain the main differences between using FSCK and using journaling for crash recovery. Under what circumstances might you still run FSCK on a journaling file system?

FSCK (File System Check) is a utility that scans the entire file system after a crash to check for inconsistencies and repair them. FSCK examines all metadata structures, including inode counts, block allocation maps, directory entries, and link counts, to detect and fix inconsistencies such as orphaned inodes, incorrect block counts, or corrupted directory structures. The main disadvantage of FSCK is that it is slow, as it must scan all metadata in the file system, and its run-time is proportional to the size of the file system. For large file systems, FSCK can take hours to complete, during which time the file system is unavailable.

Journaling is a technique that records metadata changes in a journal (log) before applying them to the actual file system structures. When a metadata operation is performed, it is first written to the journal, then applied to the file system. On a crash, the file system can recover by replaying the journal entries to restore the file system to a consistent state. Journaling provides fast recovery because only recent changes (those in the journal) need to be replayed, rather than scanning the entire file system. Recovery time is typically seconds or minutes rather than hours.

You might still run FSCK on a journaling file system under several circumstances. First, if the journal itself becomes corrupted or incomplete, the journaling recovery mechanism cannot function, requiring FSCK to repair the file system. Second, if file system corruption occurs beyond the scope of what the journal covers (such as corruption in areas not tracked by the journal), FSCK may be needed to detect and repair these issues. Third, periodic full checks using FSCK are recommended for long-term consistency, as journaling only ensures consistency of recent operations and may not catch all types of corruption that can accumulate over time. Fourth, after hardware failures that may have caused data corruption, FSCK can perform a comprehensive check to ensure file system integrity beyond what journal replay can guarantee.

41. Which of the following statements are true? **Select all that apply.**

    A. In FAT-based allocation, the file's directory entry stores pointers to all its data blocks directly, so the FAT table is only used for free-space management.
    FALSE. FAT directory entry stores only starting cluster; FAT table chains clusters.

B. Unix inodes combine several direct block pointers with one or more levels of indirect pointers to efficiently support both small and large files.
TRUE. Inodes use direct + indirect pointers.

C. A unified buffer cache avoids double-caching by using a single cache for both page-based memory-mapped I/O and traditional file system I/O.
TRUE. Unified cache avoids duplication.

D. Synchronous writes always go through the buffer cache and can be delayed there for performance, as long as metadata remains consistent.
FALSE. Synchronous writes must complete immediately, not delayed.

E. In write-ahead logging, metadata changes are written to their home locations first, and only afterwards are they recorded in the journal.
FALSE. Write-ahead logging writes to journal first, then home location.

F. Because a journal log is finite, most journaling file systems treat it as a circular buffer and eventually reuse old log space.
TRUE. Journal is circular buffer, reuses space.

Correct answers: B, C, F.

42. Match each file allocation method with the description that best fits its behavior and trade-offs.

A. Contiguous allocation	**1.** The file system records only a starting position and a length for each file. Accessing the k-th block is very fast once the start is known, but accommodating file growth may require relocating the file or adding special "extension" regions, and free space can become fragmented over time.
B. Linked allocation	**2.** Each block used by a file contains extra metadata that tells the system where to find another block of the same file. Files can easily grow one block at a time without moving existing data, and free space can be used in small pieces, but jumping directly to the k-th block may require visiting many other blocks first.
C. Indexed allocation	**3.** Information about where a file's blocks live is collected in a separate structure that stores many block references together. This allows the system to compute the location of the k-th block with one or a few lookups, even if blocks are scattered, but it consumes additional space for that structure and may be wasteful for very small files.

A → 1, B → 2, C → 3.

- **A. Contiguous allocation → 1.** The file system records only a starting position and a length for each file. Accessing the k-th block is very fast once the start is known, but accommodating file growth may require relocating the file or adding special "extension" regions, and free space can become fragmented over time.

- **B. Linked allocation → 2.** Each block used by a file contains extra metadata that tells the system where to find another block of the same file. Files can easily grow one block at a time without moving existing data, and free space can be used in small pieces, but jumping directly to the k-th block may require visiting many other blocks first.
- **C. Indexed allocation → 3.** Information about where a file's blocks live is collected in a separate structure that stores many block references together. This allows the system to compute the location of the k-th block with one or a few lookups, even if blocks are scattered, but it consumes additional space for that structure and may be wasteful for very small files.

43. Consider a file system that uses inodes to represent files. Disk blocks are 2 KB in size, and a pointer to a disk block requires 4 bytes. Each inode contains 6 direct block pointers, as well as one single-indirect and one double-indirect block pointer. A *single-indirect* pointer refers to a block that contains only block pointers, and a *double-indirect* pointer refers to a block that contains pointers to such single-indirect blocks.

    What is the maximum size of a file that can be stored in this file system?

    Given:

    - Block size = 2 KB = 2048 bytes
    - Pointer size = 4 bytes
    - Direct blocks: 6
    - Single-indirect: 1
    - Double-indirect: 1

    Step 1: Calculate entries per indirect block

    - Entries per indirect block = 2048 / 4 = 512 entries

    Step 2: Direct blocks

    - Direct blocks: 6 blocks = 6 × 2048 = 12,288 bytes

    Step 3: Single-indirect

    - 1 block containing 512 pointers $\implies$ 512 data blocks
    - Single-indirect capacity = 512 × 2048 = 1,048,576 bytes

    Step 4: Double-indirect

    - 1 block containing 512 pointers to single-indirect blocks
    - Each single-indirect block points to 512 data blocks
    - Total data blocks = 512 × 512 = 262,144 data blocks
    - Double-indirect capacity = 262,144 × 2048 = 536,870,912 bytes

    Step 5: Total maximum file size

$$\begin{aligned}\text{Total} &= 12,288 + 1,048,576 + 536,870,912 \\ &= 537,931,776 \text{ bytes} \\ &= 512.75 \text{ MB} \\ &\approx 513 \text{ MB}\end{aligned}$$